Exploiting Criticality to Reduce Branch Misprediction Penalties

Tim Lee and Jim Chen
Adviser: John Kubiatowicz
CS252 Fall 2003

Abstract

Conditional control instructions (branches) have long been a bottleneck for microprocessors. To overcome this bottleneck architects are now predicting the direction and target of branches. However, while years of research has led to increasing accuracy of branch prediction, the penalty of a branch misprediction has been increasing do to increasing pipeline depths. In this paper, we attempt to use a small cache to reduce the penalty of branch mispredictions, allowing for more aggressive pipelining and clock cycle scaling.

Based on Brian Fields and Ras Bodik's previous work on criticality, we conduct an in-depth analysis on branch mispredictions' impact on the performance of a modern microprocessor. We introduce a criteria called "critical D chains" that identifies the branches that will be most harmful if mispredicted. Once the most important branches are identified, we reduce the branch misprediction penalty by using a small trace cache and using predicted criticality as the basis for our replacement policy. Our cache resulted in small, but noticeable gains, leading us to believe that with more engineering work, the concept of long critical D chains can be used to effectively decrease the affect of branch mispredictions on overall IPC (Instructions per Cycle).

1. Introduction

Most modern processors are constructed based on the idea of out-of-order and multi-stage pipelines. One direction for improving performance is to increase the number of stages of the pipelines and reduce the cpu clock period. The biggest problem with this modification is the penalty of branch misprediction. When a branch is mispredicted, the processor must to flush the pipeline and refill the pipeline with newly fetched instructions. The time it takes for the pipeline to be filled again is proportional to the length of the pipeline.

Here we propose a design that can reduce the penalty of branch mispredictions. The design incorporates the idea of criticality and a trace cache. We use a the concept of a critical D chain to determine the "importance" of a branch, and try to exploit the branches that are most costly once mispredicted.

This paper will first discuss some background information about criticality and branch mispredictions and how they apply to our study. Then, we propose an idea of how to take advantage of criticality, and finally we show and discuss results of a simulated model of our structure.

2. Criticality

Our overview of criticality can be divided into three main parts: critical paths, critical nodes, and critical chains. The concept of critical paths and critical nodes are discussed in Ras Bodik and Brian Fields' previous work [1]. On top of that we present the idea of critical chains.

2.1 Critical Paths

To define critical paths, we first divide an instruction up into different parts corresponding to the different stages that an instruction goes through in a processor pipeline. The granularity of any criticality model is determined by how many and what stages are chosen in this division. For our purposes, the most basic model was used, which divides an instruction into decode, execute, and commit. Each part is then considered as a node in a graph; so each instruction consists of three nodes and a program is a graph with number of nodes equal to three times the number of instructions executed. Dependencies exist between different nodes. For instance, obviously all the commit nodes of the instructions will depend on the execute nodes of the same instructions, and the execute nodes will depend on the decode nodes. All the decode and commit nodes will depend on the previous ones if we issue in order and have a reorder buffer. Furthermore, if there are data dependencies between the instructions, a decode or execute node may depend on a previous execute node.

Here we represent the dependencies between nodes by drawing an arrow from one node to another node that depends on it. The relationship will form a directed acyclic graph with weighted edges (corresponding to the latency between the two nodes), with the decode node of the first instruction as the starting node and the commit node of the last instruction as the ending node.

With the dependency graph constructed, we can determine whether or not an edge is critical. An edge is defined to be non-critical if the overall run time stays the same while we reduce the weight on that edge. An edge is defined to be critical if it is not a non-critical edge. The critical path of a program is formed by following the edges that are critical.

2.2 Critical Nodes

Critical nodes are defined merely for convenience. A node is defined to be critical if it is part of the critical path. This notation is useful because we can use the technique of token passing along *last-arriving* edges described in [1] to try to estimate the critical path. Obviously, an edge is not part of the critical path if it is not the last arriving edge of a node; we can decrease the weight of such edges and there would not be any performance gain because the node still has to stall until the last edge arrives.

If we trace through all the *last-arriving* edges, we form an estimate of the actual critical path. Of course, this path may not be entirely correct, but this is a simple way to approximate the critical path. It would be unfeasible to compute the actual critical path using hardware at run time. The estimation using last arriving edges, on the other hand, can be relatively easily computed by hardware.

2.3 Critical Chains

With critical paths and critical nodes defined, we started looking into critical chains. We define a critical chain as follows: a set of nodes after a branch is considered to be a critical chain if they form part of the critical path. We decided to examine the lengths of critical chains after mispredicted branches based on the intuition that nodes after a branch are likely to be critical because of the pipeline flush.

2.4 Critical D Chains

From our experiments (see Figure 2), it is clear that critical chains after a branch mostly consist of decode nodes. This is not surprising since the processor would flush the whole pipeline when a branch is mispredicted. After the pipeline is flushed there would be nothing left to execute and commit, and therefore the chain will be made of a list of decode nodes, which we define as a critical D (decode) chain.

For the interest of our study, long critical D chains are better since there is

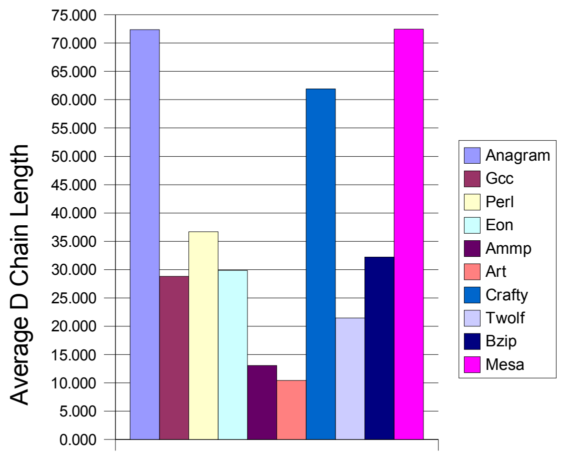

Figure 1. Mean Critical D Chain Lengths

more room for exploiting. A long critical D chain means that we could potentially speed up the program more by caching the instructions that are critical. Depending on how much information we store in our cache, it is possible to remove all the latency for the front end of the pipeline for each instruction in the cache.

2.4.1 Computing Critical D Chains

Using the lengths of critical D chains in any structure in an actual microprocessor would require an efficient hardware structure to compute the lengths of these chains. A first requirement is that the base architecture must include a criticality predictor, such as the one described in [1] based on token passing along *last-arriving* edges. Once this is in place, it may need to be extended to track D-node criticality. For example, to modify the token passing predictor, one would need to plant tokens at D nodes to find the D criticality since the proposed scheme finds only E criticality, planting tokens in E nodes.

Once the criticality for each node can be predicted, there is a counter for each outstanding branch (at most one counter per Reorder-Buffer entry) that is incremented on each consecutive instruction that is D critical. The information in this counter must then be retained somewhere. Section 4 describes how we decided to retain the information in these counters.

2.5 Predictability

To take advantage of long critical D Chains, it may be important to be able to predict the length for a particular branch. Thus, we took measurements for the standard deviation of each branch away from the mean (see Figure 2). Unfortunately it seems that many of the benchmarks exhibit extremely poor variance, possibly to the point of this attribute being memoryless.

Test	***Mean***	***Standard Dev.***			
		Hist 0	***Hist 8***	***Hist 16***	***Hist 24***
Ana-gram	72.44	35.69	35.52	34.98	34.20
Gcc	28.85	33.16	31.05	29.03	27.03
Perl	36.68	30.45	27.97	24.57	21.51
Eon	29.83	28.34	17.36	15.66	13.82
Twolf	21.45	23.91	22.96	22.09	20.90
Bzip	32.20	41.96	41.64	41.09	40.15
Mesa	72.47	41.84	40.82	37.84	36.17

Figure 2. Means and Standard Deviations for Various Tests, Hist x indicates pattern history of length x

We also ran tests using branch pattern history using a branch history register with each information associated with both a particular branch and the history register. The results showed that using pattern history does help with predictability, with extra branch history register bits improving the variance in a linear fashion, but the variances of several benchmarks are still high relative to the mean. We conducted the rest of our experiments using no history because the extra hardware spent on keeping pattern history seems to outweigh the minor gains in prediction accuracy. Nevertheless, it may still be possible to get some performance gain simply from the knowledge that the average (mean) length of the chains are frequently quite long as we will see in later sections.

3. Branch Mispredictions

Our work is predicated on the idea that to improve performance, architects will continue the current trend of increasing pipeline lengths and lowering cycle times. Recent work by Sprangle and Carmean from Intel indicates that a Pentium® 4 processor like architecture would continue to see performance improvements until the branch misprediction pipeline reaches about 52 stages [2]. At that point, the decrease in IPC due to branch mispredictions, makes any further increase in frequency/pipeline length counter-productive. Their work showed that the per clock sensitivity of IPC due to increases branch misprediction latency is not as high as for other pipelines such as the ALU and cache pipelines. However, since the length of the branch misprediction pipeline is very long, it has the most impact on overall IPC. So in our work, we set out to use the ability to predict criticality and in particular, the ability to compute the length of decode-critical chains after branch mispredictions to reduce the impact of branch mispredictions on IPC. We varied branch misprediction penalties up to 50 to verify that our simulator also showed similar IPC sensitivity (Figure 3).

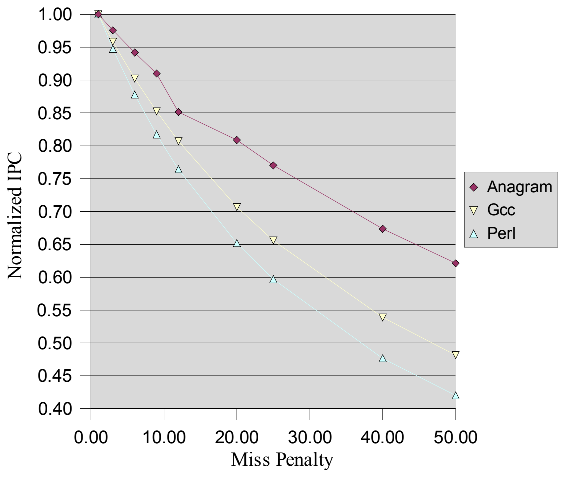

Figure 3: Sensitivity of IPC to Miss Penal-

3.1 Branch Prediction Accuracy

We first attempted to see if the decode-critical chains could be used to help branch accuracy, since this would reduce the number of times that the long branch misprediction pipeline would be used. Figure 4 shows the plot of branch accuracy vs average critical decode chain length. This plot shows very little correlation between the two, leading to the conclusion the critical decode chains probably cannot be used to increase branch accuracy.

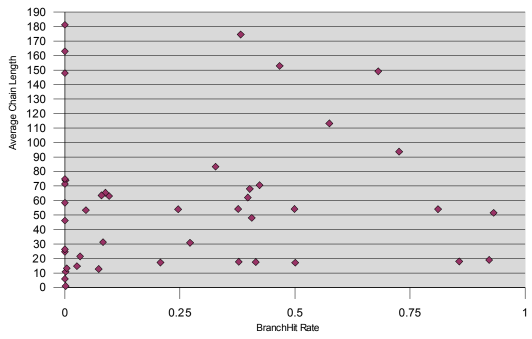

Figure 4. Branch Accuracy vs Average Critical Chain Length

3.2 Branch Misprediction Penalty

Since there was little evidence to indicate that we should pursue branch accuracy, reducing the branch penalty was the other target. Long branch misprediction penalties come from the fact that the back-end of the microprocessor must be flushed and new instructions are re-fetched, decoded, renamed, etc on any branch misprediction. So if we could cache the results of one or more of these front-end processes, the execution core would be reached more quickly the penalty would be reduced. The length of the critical decode chain can be viewed as an indication of how many instructions are bottlenecked by the ability of the front-end to feed the execution core. For example, if the critical decode chain is 24 instructions long on an architecture that has a throughput of 4 instructions per cycle, 6 cycles worth of instructions are bottlenecked by the front-end.

3.3 Trace Cache

One mechanism used to reduce both branch misprediction penalty and to increase branch prediction accuracy is a trace cache. The idea behind a trace cache is that there are some sequences of instructions (that may or may not run across conditional control instructions) that are executed frequently. So this trace cache stores entire sequences of instructions. Storing traces allows for more sophisticated branch prediction since subtle correlations between branches may be captured by a trace cache, but not by a traditional branch predictor. If the contents of the cache are pre-decoded instructions, then the decode latency is reduced to looking up and entry in the cache. This is the aspect of the trace cache that may be used to reduce the branch misprediction penalty.

We propose using a small trace cache with predecoded instructions to reduce the branch misprediction penalty. Since predecoded instructions get larger than plain instructions, this cache must have few entries to keep the overall size and and access time low. This cache is especially important after branch mispredictions because of the criticality of the front-end after branches. Furthermore, the length of the critical decode chain after a particular branch will affect how important the cached trace is. Longer critical decode chains will have more instructions sped up (finished earlier) by the trace cache than chains that aren't as long (assuming that the rest of the trace selection mechanism is unchanged). The following section describes the details of our cache system.

4. Our Cache

This section describes the main parts of our cache, the Auxiliary Branch Information Buffer, and the Branch Misprediction Trace Cache, and explains how they function in a typical modern microarchitecture.

4.1 ABIB

The Auxiliary Branch Information Buffer (ABIB) stores information about branches to be used by other structures. In this paper, we consider it as an abstract cache-like object which can have a variety of associativities, sizes and replacement policies. The input to the ABIB is an address and some information, and a read/write signal. On a read, the ABIB uses the input address as an index into the cache and outputs the information at that location. On writes, it takes the input information and writes it to the index associated with the address. Some concrete examples of ABIB are Branch Target Buffers and tables of counters found in branch prediction schemes which store addresses and branch history information respectively. The affect of aliasing due to finite size and non-full associativity in ABIB structures have been studied before; so we assume an ideal, infinite, fully-associative ABIB in our work to concentrate on the behavior of the BMTC (section 4.2). The types of information we stored in our simulations were current mean of the length of the critical D chain, the total number of

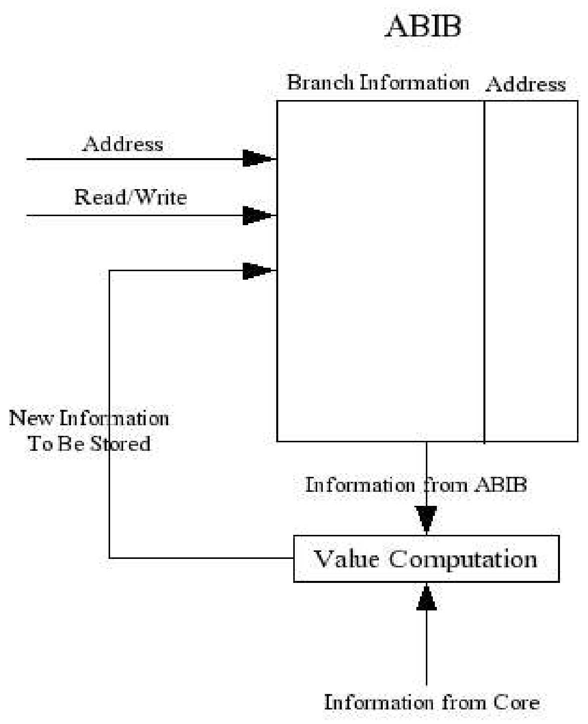

Figure 6. Block Diagram of Auxiliary Branch Information Buffer

branches mispredicted (or another measure of the frequency of this branch relative to others), and traces for each branch. The mean can be computed on the fly using an adder, multiplier and divider, although this will be an approximation in hardware due to finite counter sizes. We store the trace because this structure also serves as the source for traces that enter the BMTC when evictions occur.

4.2 BMTC

The Branch Misprediction Trace Cache (BMTC) is the small, fully-associative cache that stores traces with predecoded instructions. It also stores in each entry, a value which is computed from the information stored in the ABIB. This value field is used to determine which entry will be evicted when necessary, and when it is necessary to do so. One of our simulation parameters is how to compute this value.

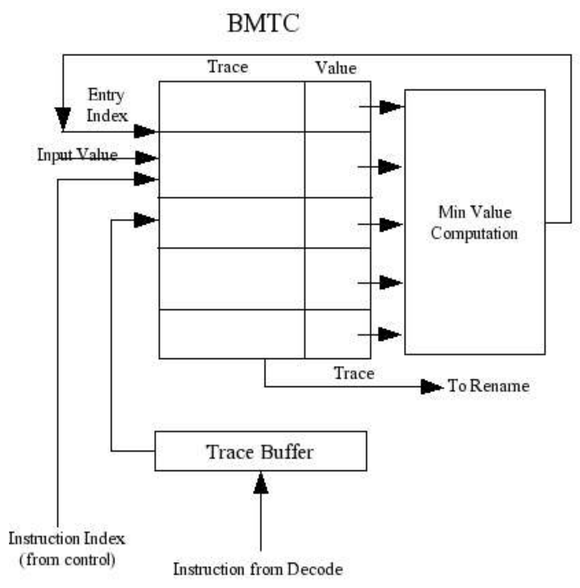

Figure 7. Block Diagram of Branch Mispredict Trace Cache

The inputs to the BMTC are an two indices, a trace, and a read/write signal, which function similarly to the inputs of the ABIB. Two indices are required to access each trace and the individual instructions of each trace. Finally, it should be noted that the value computed can take a long time without large penalty since the most important latency is the time to lookup a trace and not the the latency to update it. If the BMTC or ABIB is updated slowly, the only penalty is a slightly less accurate value in the BMTC; so it may be worth the tradeoff to make the value as meaningful as possible (by making a pipelined complex value calculation), but making the update slower.

4.3 Interaction

If the base architecture does not already use a trace cache, a trace buffer is added which takes input from the Decode section of the processor and keeps a buffer of the current trace. This buffer is stops taking input when it becomes full, or when its data is written to the BMTC. The structure of this buffer is a circular buffer of decoded instructions with maximum size equal to the number reorder buffer entries. When a branch is sent to the

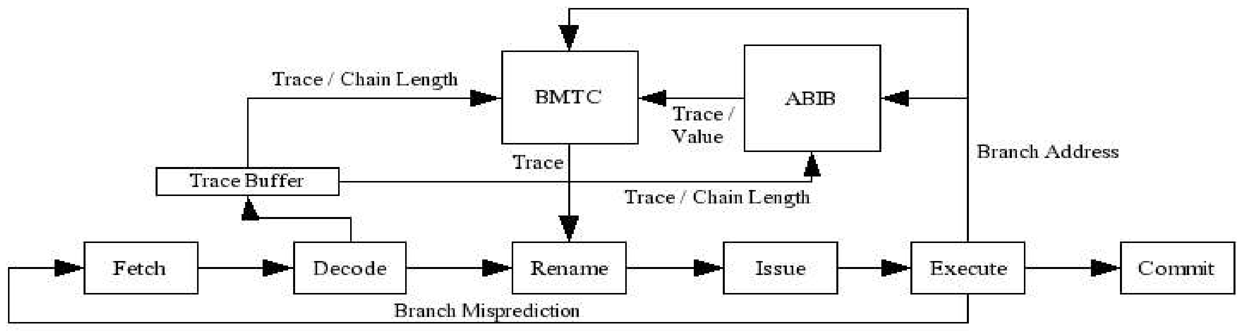

Figure 8. Pipeline with BMTC and ABIB added to shortcut branch mispredict loops

reorder buffer, a pointer to the current position in the buffer is saved for that branch (this can be saved in the ROB). When a branch is mispredicted, the trace along with the length of the critical D chain (computed as a trace is being sequenced) is copied into the BMTC and/or ABIB when the branch hits the end of the ROB.

If the base architecture does already use a trace cache, the buffer stores micro-ops directly from the primary trace cache as if it were coming from the Decode section of the processor. For our simulations, we did not assume this type of architecture, but our cache could still be effective if it used more aggressive caching (see future work). It is also possible that the replacement policy used in our cache would be more effective at keeping the trace in our cache over the base trace cache.

As instructions commit and reveal branch mispredictions, the length of the critical decode chain for the current mispredicted branch is sent to the ABIB to update its information; the updated information is computed using the previous information for this branch and the new information from the current branch. This allows a variety of schemes for using branch information history in our replacement policy. The branch misprediction also triggers a lookup in the BMTC to see if the current misprediction is in the cache. If it is, the trace is used and the input to the Rename unit is switched from the Decoder to the BMTC. Also, the value in the BMTC is updated by the newly computed information for the ABIB. If it is not in the trace, the regular branch misprediction pipeline is used. We explored two ways of updating the BMTC. The first is to use the information that is being stored in the ABIB to immediately compute a new value and compare it against the weakest (what this means depends on the value scheme) value in the BMTC. If the new value is better than the old one, an eviction takes place (Figure 7).

Another method is to keep the current best value computed from the information in the ABIB (but not in the BMTC), in a register and update this if necessary when we update the ABIB. Then, whenever a value is updated in the BMTC, it is checked against this value and an eviction takes place if necessary. This method would save the computation of finding the weakest value in the the BMTC, but does not maintain the invariant of keeping the highest k values in the cache. Simulations showed that gains using this method are must worse than using the other method, so we used the first one.

5. Simulations

We evaluated our cache using the open-source, SimpleScalar toolset originally developed by Todd Austin at University of Wisconsin, now maintained and developed by SimpleScalar LLC (www.simplescalar.com). We also used the criticality package developed by Brian Fields and described in [1]. We chose these tools simply due to their easy availability and ease of modification. Although SimpleScalar was not developed to simulate extremely long pipelines, we are using it only to get a rough estimate of possible gains and if warranted, further study can be conducted using a more detailed simulator/cache model.

5.1 Simulation Methodology

Our baseline SimpleScalar simulation configuration was as follows:

- 4-way SuperScalar
- Out-of-order Execution Core
  - 4 Integer ALUs, 4 FP ALUs
  - 1 Integer Multiplier/Divider, 1 FP Multiplier/Divider
  - 128 Entry Reorder Buffer
- 64KB 2-way set associative Split L1 Cache
  - 1 cycle hit to I-Cache
  - 2 cycle hit to D-Cache
- 256KB 2-way set associative Unified L2 Cache
  - 12 cycle hit
- 8 byte bus to DRAM
  - 100 Cycle latency for first word from DRAM
  - 2 cycles for rest of the words in a line
- 4KB 4-way set associative Branch Target Buffer
- 16KB G-Share branch predictor
- 50 cycle branch misprediction penalty
- All tests run for a max of 25 million instructions
- The criticality tracer from [1] was modified to determine the D criticality for all instructions

We simulated the trace cache by reducing the branch misprediction penalty linearly with the number of decode critical instructions in the trace, with a lower limit of 25 cycles. Any non-decode critical instructions would not benefit from early decoding, so their penalty remains unchanged. Using this method, we cannot get exact numbers about the size of the our cache, but we can make a rough estimate as follows: a trace cannot be longer than 100 instructions (a limit we set in simulation) and each instruction in the trace cache takes up 8 bytes (since decoding instructions expands them into a less dense encoding, more conducive to use by the processor's datapath) and we have a 5 entry BMTC = 4000 bytes. This also ignores the size of the ABIB, but this structure can be far away from the execution core and thus can be quite big and slow.

Although we managed to estimate the performance of our design by running simulations in SimpleScalar, future studies should be conducted on another simulator that can correctly and accurately reflect the performance of our cache model on a processor with very deep pipelines.

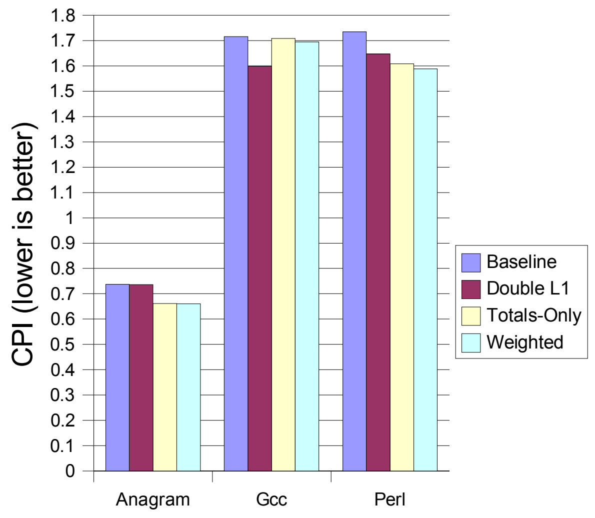

Figure 8.
Results of 5 Entry BMTC

5.2 Results

Figures 8 and 9 show the gains achieved by our cache scheme with various ways of computing the "value" compared to the baseline configuration, and compared to increasing the size of the L1 cache by 64K, an alternate way of spending extra transistors to try to increase performance. Using just the means of the length of the critical D chains as our value showed almost no improvement. We attributed this to low frequency with which those particular branches occurred. Thus, to remove those outliers, we tried to weight the mean of each branch by the number of times that branch was seen (indicated by the "Weighted" bars in the graph). This was also compared to just using the total number of branches that the time was seen as the value (indicated by the "Totals-Only" bars in the graph).

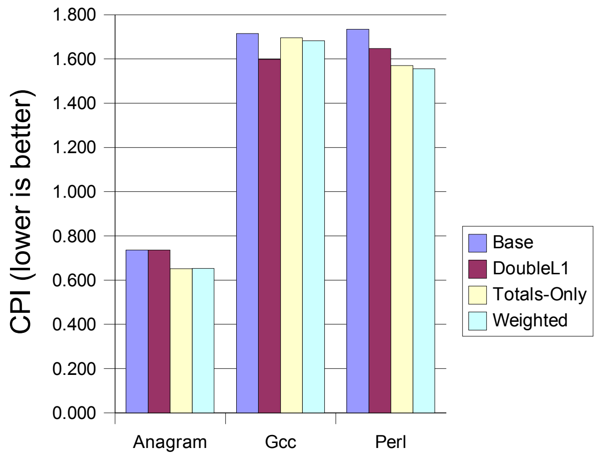

Figure 9.
Result of 10 Entry BMTC

5.3 Analysis

Our results did not perform quite as well as we had hoped. The weighted cache appears to better than a doubled-size L1 cache except for gcc, and performs just slightly better than totals-only cache. Considering that the size of a 5-entry weighted cache (estimated at 4000KB) is much smaller than doubling the L1 cache (64KB), it performs quite well compared to double L1 cache. However, a equal-size totals-only cache also performs almost as well. This result is not complete surprising considering the maximum critical D chain we observed in our benchmarks is no more than five hundred, while the number of misses for a branch ranged anywhere from a thousand times to thirty thousand. Because the number of misses for a branch is much greater than the critical D chain length and is has a wider range, the product will depend largely on the number of misses. Even with the overwhelming effect of the number of mispredictions, we observed slight performance gain by using the weighted cache, which is encouraging. Due to the lack of time and resources, we were unable to refine the design of the weighted cache. However, increasing the weight of the critical D chain length seems to be the next logical approach and may yield better results.

The 10-entry cache, while twice as big, does not achieve nearly twice the gain as the 5-entry one. We speculate that the 5-entry cache already obtains most of the gains possible with this technique. Thus, although it may be a good solution in terms of performance gain per KB of low latency storage, it does not scale well and should thus be thought of as a small optimization.

While we were examining the correlation between average chain length and branch accuracy, we discovered that there exists a correlation between number of times a branch is mispredicted and the accuracy of the branch. Branches that are mispredicted more than a constant number of times (this constant varies per benchmark) have a much lower accuracy compared to the other branches. Therefore, one may argue that the performance gain we obtained with our cache is simply a result of caching the branches that are mispredicted most often. Indeed, the weighted cache has limited improvements compared to the totals-only cache, but we must consider the fact that number of branch misses are often on the order of thousands or tens of thousands, whereas the length of critical D chains are on the order of hundreds. Despite the obvious fact that number of misses has an overpowering effect, there is still observable gain for using the weighted cache. A more detailed analysis of the value of the concept of critical D chain length is beyond the scope of this paper. Nevertheless, the results seem promising. We believe it is worthwhile to further look into the design of the weighted cache. There may be significant performance gain by carefully engineering of a function of number of misses and critical D chain length, which counteracts the overpowering effect of number of misses.

6. Future Work

Other than the hardware structure we describe in this paper, there are a number of ways of exploiting branch mispredictions using the idea of criticality and critical D chains after branch mispredictions: aggressive caching, switching between an out-of-order processor and an in-order processor or between a long pipeline and a short pipeline, and using reconfigurable architecture if available.

The focus of this study is exploiting the critical D chains. Therefore, with extremely aggressive caching that stores information about predecoding, pre-renaming, or even pre-issuing/pre-scheduling, it is possible to entirely eliminate the time spent on the front end of the pipeline.

It would also be helpful if we were able to more accurately predict the length of critical D chains for each branch. If we were to know the length length of a critical D chain, we can allocate appropriate portion of the memory and avoiding wasting cache space. As our simulation suggests, the average lengths of critical D chains are all reasonably long. However, there does not seem to exist a clear correlation between the average critical D chain lengths and their variance, which means a naïve way of predicting the length based on the mean computed on the fly would not work. Judging from the data obtained from our simulations, there also seem to be very little correlation between average critical D chain length and branch accuracy. Perhaps a close examination on path and pattern history would reveal a predictable pattern of the length of the critical D chains.

The concept of critical chains can also be applied to save power consumption. When we are in the midst of executing a critical D chain, out-of-order execution would not be beneficial since all instructions depend on previous instructions. In that case switching to an in-order processor would not result in a performance loss, and the overall power consumption may reduce because we could skip some of the required checks for out-of-order execution and perform less computations. Assuming we can effectively predict the length of critical D chains, we may be able to find chains that are long enough such that switching to an in-order processor would reduce the overall power usage.

Switching to a shorter pipeline when a branch followed by a long critical D chain is about to be executed may also reduce the overall run time since it would take less time to refill a shorter pipeline. Presumably the shorter processor would run at a slower speed, and we would switch back to the long pipeline once the pipelines are filled. However, without an in-depth study, it is uncertain to us when and how the switching between the pipelines would be performed.

Finally, if there are reconfigurable structures that can function as both a functional unit and a fetch or decode unit, we could allocate more resources for decoding and fetching when a long critical D chain is coming up. With extra decode and fetch units, the front end of the instructions would be completed faster, and we would be able to reduce the time spent on the critical D chain.

Unfortunately, such structures are unknown to us.

7. Conclusion

In this paper, we introduced the idea of a critical D chain based on previous work in criticality of instructions [1]. We then found that there exist many long critical D chains after mispredicted branches. This information then lead to our proposal of a cache structure that incorporates a traditional cache (ABIB) and a small trace cache (BMTC) with the replacement policy as a function of the predicted critical D chain length. The results showed that this structure, using an estimated 4KB of low latency memory, had small, but observable gains, even compared to a configuration that added 64 of L1 cache. Thus we see that there is some merit in the ability to predict the length of critical D chains.

8. References

[1] Brian Fields, Shai Rubin and Rastislav Bodik. "Focusing Processor Policies via Critical-Path Prediction." The 28th International Symposium on Computer Architecture, 2001.

[2] Sprangle and Carmean. "Increasing Processor Performance by Implementing Deeper Pipelines." Proceedings of the 29th Annual International Symposium on Computer Architecture, 2002.

[3] E. Rotenberg et. al. "Trace Processors." Proceedings of International Symposium on Microarchitecture (MICRO-30), 1997.